Translating C to Safer Rust – Extended Version

MEHMET EMRE, University of California Santa Barbara, USA
RYAN SCHROEDER, University of California Santa Barbara, USA
KYLE DEWEY, California State University Northridge, USA
BEN HARDEKOPF, University of California Santa Barbara, USA

Rust is a relatively new programming language that targets efficient and safe systems-level applications. It includes a sophisticated type system that allows for provable memory- and thread-safety, and is explicitly designed to take the place of unsafe languages such as C and C++ in the coding ecosystem. There is a large existing C and C++ codebase (many of which have been affected by bugs and security vulnerabilities due to unsafety) that would benefit from being rewritten in Rust to remove an entire class of potential bugs. However, porting these applications to Rust manually is a daunting task.

In this paper we investigate the problem of automatically translating C programs into *safer* Rust programs—that is, Rust programs that improve on the safety guarantees of the original C programs. We conduct an in-depth study into the underlying causes of unsafety in translated programs and the relative impact of fixing each cause. We also describe a novel technique for automatically removing a particular cause of unsafety and evaluate its effectiveness and impact. This paper presents the first empirical study of unsafety in *translated* Rust programs (as opposed to programs originally written in Rust) and also the first technique for automatically removing causes of unsafety in translated Rust programs.

Additional Key Words and Phrases: Rust, C, Automatic Translation, Memory-Safety, Empirical Study

1 INTRODUCTION

Rust is a relatively recent programming language designed for building safe and efficient low-level software [Klabnik and Nichols 2018]. It provides strong static guarantees about memory and thread safety while avoiding the need for garbage collection, and allows for low-level data manipulations often required by system-level software. Rust has been used for building operating systems, web browsers, and garbage collectors [Anderson et al. 2015; Levy et al. 2015; Lin et al. 2016] and it is being adopted into complex software projects with large C/C++ code-bases such as Firefox [Bryant 2016], the Linux kernel [rus [n.d.]a,n], and Android [Stoep and Hines 2021].

An alarming amount of critical systems software (much of which predates the development of Rust) is instead written in unsafe languages such as C and C++. Those languages' lack of memory and thread safety has led to numerous critical bugs and security flaws [noa 2021a,b; Durumeric et al. 2014] with attendant costs in terms of both money and human lives [Durumeric et al. 2014]. In light of Rust's recent development and promise of safety, a natural question arises about the possible benefits of porting software from these unsafe languages to Rust, eliminating a large class of potential errors. In fact, there has been some informal investigation into the question of how effective Rust would be at fixing critical errors in existing C code (after all, not all bugs and security

Authors' addresses: Mehmet Emre, emre@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Ryan Schroeder, rschroeder@ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Kyle Dewey, kyle.dewey@csun.edu, California State University Northridge, Northridge, CA, USA; Ben Hardekopf, benh@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA.

XXXX-XXXX/2021/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

flaws are due to memory or thread unsafety). As an example, one indicative (though unscientific) study done on cURL, a popular data transfer utility written in C, conservatively estimates that using Rust would eliminate 53 of the 95 known cURL security flaws [Hutt 2021].

One obvious objection to porting existing software into Rust is the sheer effort required to rewrite the code in a new language. An automated, rather than manual, translation would make that effort much more practical. The primary barrier to such an automated translation is Rust's sophisticated type system which it uses to provide the desired memory and thread safety guarantees. To produce verifiably safe Rust code from unsafe C code, for example, requires the translator to analyze the relevant properties of the C code and create a suitable well-typed Rust program that correctly expresses those properties.

There have been several industry-backed attempts to automatically translate C programs to Rust [Citrus Developers [n.d.]; Immunant inc. 2020b; Sharp 2020]. These translations are purely syntactic in nature, producing memory- and thread-unsafe Rust code that closely mimics the original C code and explicitly bypasses the safety checks of the Rust compiler (by marking all translated code with Rust's built-in `unsafe` annotation). While these tools provide a good starting point for automated translation, they leave the hard work of manually reasoning about the safety properties of the translated program and rewriting the code to enable the Rust compiler to verify those properties to the developer. To our knowledge there has been no academic or industry investigation into the question of whether and how unsafe languages can be automatically translated into *safe* Rust programs. This paper makes two major contributions towards the goal of automatically translating sequential C programs (for this stage of the work) to *safer* Rust programs, i.e., the goal for now is not complete safety but simply more safety than the existing naive syntactic translations.

Our first contribution is a quantitative study on the sources and causes of unsafety present in Rust programs that have been syntactically translated from C programs. While there have been studies on unsafe code in native, hand-written Rust programs [Astrauskas et al. 2020; Qin et al. 2020], this is the first study that examines automatically translated Rust programs. We focus on Rust code translated from C using the existing `c2rust` translator [Immunant inc. 2020b]. Our findings indicate that unsafety in automatically translated Rust code differs in various significant ways from unsafety in natively written Rust code. For example, a prevalent source of unsafety in automatically translated code, unlike native code, is due to the use of *raw pointers*: the translation to Rust converts all C pointers into raw pointers, and any dereference of a raw pointer must be marked as unsafe. We break down all of the sources of unsafety present in our translated benchmarks, quantify how often they occur, explain what causes these sources of unsafety in the original C programs, and quantify the impact of addressing each source of unsafety on the overall safety of the translated Rust programs.

Our second contribution, informed by our study, is a technique for automatically generating safer Rust code by addressing one common cause of unsafety. We focus specifically on the use of raw pointers in the translated programs. Idiomatic Rust code instead uses safe *references* with explicitly annotated lifetime information that allows the Rust compiler to safely deallocate the associated memory when it is no longer needed. Rust uses an ownership-based model for statically reasoning about references, shared references, mutability, lifetimes, and overall memory- and thread-safety [Boyapati et al. 2002, 2003]. Using this model, valid Rust programs are automatically proven safe via Rust's *borrow checker*. Invalid Rust programs, i.e., those unable to be statically proven as safe, are rejected by the borrow checker (which ignores code explicitly marked as unsafe). We make the key insight that we can piggyback on Rust's borrow checker in order to extract the lifetime, sharing, and mutability information we need to turn a subset of raw pointers into safe references. We introduce and implement a translation technique based on this insight which takes naively translated, completely unsafe Rust programs and generates *safer* Rust programs (specifically, in

Table 1. Benchmark programs, ordered by Rust lines of code. The benchmarks that come from the `c2rust` manual are marked with **bold**. LoC = lines of code, not counting comments or blank lines. The tulipindicators and robotfindskitten benchmarks are abbreviated as TI and RFK, respectively.

Benchmark	Application Domain	C LoC	Rust LoC	# Functions	# `unsafe` Functions
libcsv	Text I/O	1,035	951	23	23
urlparser	Parsing	440	1,114	22	21
RFK	Video games	838	1,415	18	17
genann	Neural nets	642	2,119	32	27
lil	Interpreters	3,555	5,367	160	159
json-c	Parsing	6,933	8,430	178	178
libzahl	Big integers	5,743	10,896	230	230
bzip2	Compression	5,831	14,011	128	120
TI	Time series analysis	4,643	19,910	234	229
tinycc	Compilers	46,878	62,569	662	625
optipng	Image processing	87,768	93,194	576	572
libxml2	Parsing	201,695	430,243	3,029	3,009
Total	—	366,001	650,219	5,292	5,210

this case, one with fewer raw pointers). We evaluate our implementation on a set of C benchmarks and report on its effectiveness.

The specific contributions of this paper are as follows:

- A study of the sources of unsafety in Rust code that has been produced by `c2rust` (Section 2);
- A technique to rewrite a particular source of unsafety in translated programs (a specific kind of raw pointer) that hooks into the Rust compiler to extract type- and borrow-checker results and uses them to generate verifiably safe code (Section 3);
- An implementation of this technique[1] with a corresponding evaluation of its effectiveness (Section 5).

We end with a discussion of related work (Section 6) and conclusion (Section 7).

2 UNSAFETY IN TRANSLATED RUST PROGRAMS

We investigate the various sources of unsafety in Rust programs that have been translated from C using `c2rust`. While there are existing studies of unsafe code in the native Rust ecosystem [Astrauskas et al. 2020; Qin et al. 2020] our investigation is specifically about automatically translated Rust programs, which may have a different distribution of unsafe code than Rust programs written by developers.

2.1 Benchmarks

Previous studies of unsafe Rust code have taken advantage of large repositories of native Rust programs such as `crates.io`. There does not exist a large repository of Rust code that has been translated from C, and so we must collect our own benchmark suite. While there are many existing C programs to choose from, each translation requires a fair amount of manual labor to correctly insert `c2rust` in that C program's particular build process, and also `c2rust` itself does not work on all C programs and build environments.

We have collected 12 open source C benchmarks of various sizes and application domains, as shown in Table 1. Six of the benchmarks came from the `c2rust` manual [Immunant inc. 2020a] (marked with **bold** in the table); the remaining six came from GitHub. We picked benchmarks from

[1]We will submit our implementation for artifact evaluation.

a variety of application domains, as described in the table. Table 1 shows that, on average, the translated Rust programs are 1.8× larger than their C counterparts. Decreases in translated LoC arise because `c2rust` removes obviously dead or unreachable code. Increases in translated LoC come from macro expansion, adding function declarations for functions included from the headers, translation of increment and decrement operators[2], and annotations such as `#[no_mangle]` and `#[repr(C)]` to make the Rust code compatible with the C ecosystem.

Table 1 also shows that the vast majority of functions in the translated code are marked `unsafe`. Specifically, all translated functions directly from the original C program are marked `unsafe`, and only auxiliary functions generated and introduced during the translation itself are marked safe. Although all functions directly coming from C are conservatively marked `unsafe` by the translation, we observe that some do not actually require the `unsafe` tag. In Section 2.2 we quantify how many functions are unnecessarily marked `unsafe` by the translation. Furthermore, we characterize different sources of `unsafe` and quantify how prevalent they are in the program.

Threats to Validity. Our benchmark suite is limited in the number of benchmark programs because of the manual effort required to: (1) convert each C program to a corresponding Rust program with necessary adjustments to their respective build processes; and (2) reorganize the code (such as unit tests) in a way that Cargo, the de-facto standard build system for Rust, can build the resulting Rust project reliably. The size of the benchmark suite means that the percentages we report may not reflect the percentages of a larger pool of C programs. We have selected benchmarks from a variety of domains in order to reflect different types of C programs and to increase the validity of our benchmark suite and our results.

Threats to Validity.. Our benchmark suite is limited in the number of benchmark programs bacause of the manual effort required to convert each C program to a corresponding Rust program with necessary adjustments to the build system, and reorganizing the code (such as unit tests) in a way that Cargo, the de-facto standard build system for Rust, can build the resulting Rust project reliably. We used C programs already used by c2rust developers to demonstrate their tool. As for our other benchmarks, we picked benchmarks from a variety of domains in order to reflect different types of C programs, and to increase the validity of our benchmark suite and our results.

2.2 Provenance of Unsafety

The Rust Reference [The Rust developers [n.d.]b] defines the following sources of unsafety:

(1) Dereferencing a raw pointer
(2) Reading from or writing to a mutable global (i.e., `static`) or external variable
(3) Reading from a field of a C-style untagged union
(4) Calling a function marked `unsafe` (including external functions and compiler intrinsics)
(5) Implementing a trait that is marked `unsafe`

These categories are too coarse-grained for our purposes. In particular, for our benchmarks Category 4 includes almost all calls to the other functions in the program because almost all functions in the program are initially marked `unsafe`. Category 4 also includes the use of inline assembly and unsafe casting, which we would like to separate from other sources of unsafety for our study.

We have refined the official categories above into distinct *features*, where each feature reflects a particular unsafe feature in Rust. These features give us a clearer picture of programs translated from C. Since our benchmarks do not implement any unsafe traits (they only implement traits that can be derived by the compiler, which are all safe), we do not consider Category 5 further.

[2]Rust does not have increment-and-return operators like ++x and assignments do not return the left-hand side, so these operators are translated into multiple statements in Rust.

Our benchmarks call external functions extensively (e.g., `malloc`), making external function calls (Category 4) a major source of unsafe function calls. We count calls to `malloc` and `free` separately from other external function calls, as we conjecture that most of the allocation-related external calls can be converted to safe memory allocation mechanisms in Rust such as `Box::new`. In our benchmarks, the only unsafe Rust standard library function called is `std::mem::transmute`, used for reinterpreting/casting a value. We exclude calls to `std::mem::transmute` when it is used for casting byte arrays to C-style character arrays (which is safe under the assumption made by `c2rust` that a character is 8 bits). The resulting features that we measure for our benchmarks are as follows, where the text in `bold` indicates the column names in our tables:

- **RawDeref**: dereferencing a raw pointer;
- **Global**: reading from, writing to, or making a reference to a mutable global (`static`) or external variable;
- **Union**: reading from a field of a C-style untagged union;
- **Allocation**: direct external function calls to `malloc` and `free`;
- **Extern**: calling an external function other than a function defined in another module in the same program,[3] `malloc`, or `free`; or making an indirect call via a function pointer;[4]
- **Cast**: unsafe casting using `std::mem::transmute`;
- **InlineAsm**: using inline assembly.

We collect our data on a function-level because (1) `c2rust` marks functions `unsafe` rather than inserting `unsafe` blocks,[5] and (2) existing work on quantifying unsafe behavior of Rust programs in general [Astrauskas et al. 2020] aggregates the relevant information on a function level because different developers may prefer to use different granularities for `unsafe` blocks.

An important omission in our categories of unsafety is that of direct calls to `unsafe` functions (i.e., the original Category 4 above). As previously mentioned, this category is not useful for our translated benchmarks because almost all function calls are to `unsafe` functions, and what we are interested in is *why* the functions are unsafe. For this reason, we count sources of unsafety differently from any existing work: a function is unsafe in relation to some category above not only if it directly contains unsafe code relevant to that category, but also if it directly or transitively calls a function that is unsafe due to that category. In other words, we count a function as unsafe for a category if executing that function can exhibit unsafe behavior relevant to that category. To calculate this, we build a call graph and propagate unsafe behavior from callees to their transitive set of callers.

For each unsafe feature, we collect the following information for our study:

(1) How many unsafe functions in the program use the `unsafe` feature, directly or transitively (i.e., how many functions need the unsafe feature)
(2) How many unsafe functions in the program use *only* this unsafe feature
(3) How many times a use of the unsafe feature appears in the program text

[3] `c2rust` uses `extern` declarations to import functions from other modules in the same program. These functions can be imported directly as non-external functions after the changes described in Section 4.1.1, so we do not count these functions as external functions in our study.
[4] An indirect call could be calling an external function, and just like an extern call the compiler can only see the function signature of the callee but not the body.
[5] Except when generating shims for the `main` function, which cannot be marked `unsafe`. These shims extract the program arguments then immediately call the `main` function from the C program.

Table 2. How many times different categories of unsafety appear in each benchmark. The meaning of each column is explained in Section 2.2.

Benchmark	Union	Global	InlineAsm	Extern	RawDeref	Cast	Alloc
libcsv	0	2	0	35	174	4	0
urlparser	0	1	0	122	60	43	55
RFK	0	127	0	86	24	0	2
genann	0	164	0	183	339	3	5
lil	0	10	0	149	1,668	11	62
json-c	101	93	0	208	1,843	17	30
libzahl	0	430	29	63	2,457	0	43
bzip2	0	700	0	422	3,764	1	14
TI	0	108	0	352	1,847	84	9
tinycc	613	2,552	0	464	5,632	31	2
optipng	82	1,361	0	812	6,062	37	43
libxml2	499	3,571	0	4,593	52,546	15	15
Total	1,295	9,119	29	7,489	76,416	246	280

(4) The total size (in lines of code) of unsafe functions that directly or transitively use the unsafe feature

To get the feature counts for item 3 in the above list, we first convert the translated Rust programs to Rust High-level IR (HIR)[6], an AST-based representation. From there, we count individual features in the HIR in the following ways:

- For pointer dereferences, we count the number of raw pointer dereference nodes
- For inline assembly, we count the number of inline assembly nodes
- For interaction with mutable or external globals, we count how many times these variables are used (read from, written to, or taken a reference of) in the source code.
- For reading from a union, we count each field access involving a union, *unless* it is immediately on the left-hand side of an assignment
- For memory allocation, external functions, and unsafe casting, we count the number of static call sites to the relevant functions

Table 2 lists how many times each source of unsafety statically appears in each benchmark. We observe that there are two sources which do not appear across many benchmarks, namely C-style unions (which appear only in larger programs) and inline assembly (which is only used in one program). Table 2 shows that the most common source of unsafety is raw pointer dereferencing, which is eight times more common than the next most common source (globals), followed closely by external function calls.

Table 3 takes a function-level approach, counting the number of functions directly or transitively affected by each category of unsafety. This information is split into functions that are uniquely affected by a single category of unsafety (under the $\exists!$ columns) and those that are affected by multiple categories of unsafety including this one (under the $\exists_{\geq_2}$ columns). The $\exists_{\geq_2}$ columns will count a function multiple times, once for each category it is affected by. Functions which were marked `unsafe` by the translation but nonetheless are devoid of unsafe behavior are totalled in the false positives (FP) column; we observe that 6% of functions fall into this category. Both Table 2

[6]HIR is used internally in the Rust compiler, and is close to initial AST obtained after expanding macros, type checking, and normalizing loops and conditionals. We chose to use HIR because it provides type information needed by our analyses and it is close to the source code.

Table 3. Number of functions affected by each category of unsafety (a function may be counted multiple times if affected by multiple categories). FP denotes false positives: functions that do not contain any unsafe behavior but are marked `unsafe` by `c2rust`. The column labels are explained in Section 2.2.

Benchmk.	Union		Global		InlineAsm		Extern		RawDeref		Cast		Alloc		FP
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	
libcsv	0	0	1	1	0	0	0	9	13	22	0	4	0	0	0
urlparser	0	0	0	14	0	0	0	20	0	17	0	1	0	19	0
RFK	0	0	0	15	0	0	1	15	0	7	0	0	0	2	1
genann	0	0	0	14	0	0	1	24	0	21	0	13	1	18	2
lil	0	0	2	73	0	0	1	134	14	148	0	52	1	100	2
json-c	0	62	10	49	0	0	4	114	24	144	0	49	1	51	11
bzip2	0	0	3	79	0	0	7	85	23	82	0	3	2	26	0
libzahl	0	0	0	115	0	111	0	114	90	230	0	0	0	110	0
TI	0	0	0	13	0	0	1	104	74	175	0	73	1	16	49
optipng	0	57	4	297	0	0	14	371	126	487	0	57	7	141	29
tinycc	0	283	5	486	0	0	8	488	57	577	0	183	1	340	30
libxml2	0	198	28	2,220	0	0	39	2,359	369	2,740	0	1,156	0	1,268	183
Total	0	600	53	3,376	0	111	76	3,837	790	4,650	0	1,591	14	2,091	307

and 3 show RawDeref, Global, and Extern to be the biggest sources of unsafe behavior, typically in that order. However, while RawDeref is heavily overrepresented in terms of sheer usage (Table 2), at the function level it compares much more closely to Global and Extern (Table 3). From the standpoint of trying to make more functions safe, this is an important observation to make, as it shows that RawDeref is not much more important than Global or Extern.

2.3 Underlying Causes of Unsafety

We now investigate the behaviors in the original C programs that lead to each category of unsafety. Some categories of causes are obvious and uninteresting: mutable globals (Global) and dynamic memory allocation (Allocation) are needed in C programs for creating long-lived objects that are accessible from different parts of the program; inline assembly (InlineAsm) is used in only one of our benchmarks (`libzahl`) for architecture-specific optimizations. We examine the remaining categories in more detail below.

2.3.1 ***Raw Pointers.*** We inspected the translated benchmarks and how they use raw pointers in detail. We recognize five distinct reasons that a benchmark might have for using a raw pointer:

- The raw pointer appears as part of the public signature of an API implemented by the benchmark. This is a common occurrence in our benchmarks because most of them (except `lil` and `RFK`) are either libraries or contain libraries.
- The raw pointer is obtained via custom memory allocation (i.e., calling `malloc`). These raw pointers could be converted to safe references if we replace `malloc` with Rust's safe memory allocation and compute suitable lifetime information for them.
- The raw pointer is obtained via a cast to or from `void*`. In all cases this reason turns out to be the result of an idiomatic C method for overcoming C's lack of generics and implementing polymorphism. These raw pointers could be converted to safe references by introducing generics or traits to implement polymorphism.
- The raw pointer is passed as an argument to, or returned from, an external function call. These raw pointers can only be converted into safe references by replacing the external call.

- The raw pointer is used in pointer arithmetic. Because arrays in C decay to pointers, this reason captures most array accesses (unless the array has a fixed size known at compile time). Rust does not allow pointer arithmetic on safe references, but these raw pointers could be converted to safe references if we can convert the pointer arithmetic into safe array slices.

In our data collection we group the first two categories above into a single category named `Lifetime` because converting these raw pointers into safe references requires computing the same information for both categories and does not involve much invasive code transformation beyond changing the pointer declarations and inserting lifetime information. Note that deriving the lifetime information is needed for making pointers safe in all categories, so `Lifetime` specifically denotes pointers that do not fall into any other category. The remaining categories are named `VoidPtr`, `ExternPtr`, and `PtrArith` respectively. For each category of raw pointer we collect the following information, using the same methodology as for Section 2.2:

(1) Number of declared pointers involved in that category (Table 4);
(2) Number of dereferences of pointers in that category that appear in the code (Table 5);
(3) Number of unsafe functions that use pointers from that category (Table 6).

A pointer may be contained in multiple categories (e.g., a pointer returned by `malloc` that undergoes pointer arithmetic and is then passed to an external function). As in Table 3, we split our counts into pointers that uniquely belong to a particular category ($\exists!$) and those that belong to that category but also others ($\exists_{\geq 2}$). Because the `Lifetime` category contains only pointers not involved in other categories we only give the $\exists!$ column for it. For counting the number of unsafe functions in Table 6 we only consider those functions for which raw pointers are the only reason for their unsafety; that is, we do not consider functions that use global variables, unsafe cast, inline assembly, or read from a C-style union. "Using" a pointer means any one of declaring (as a parameter or in the function body) or dereferencing the pointer. As a reminder, we consider a function to use a pointer either if the function does so directly, or calls (directly or transitively) a function that uses the pointer.

To determine how the pointers are being used we implemented and executed a flow-insensitive, field-based taint analysis based on Steensgaard-style pointer analysis [Steensgaard 1996] and Rust's type system [The Rust developers [n.d.]b]. We chose a flow-insensitive, equality-based analysis because all values that flow into a variable and from the variable are necessarily of the same type, and if any one of those values is used for a reason on our list then that reason forces that variable and all of the places its value flows to be a raw pointer. We consider a pointer to belong to a particular category (`Lifetime`, `VoidPtr`, `ExternPtr`, or `PtrArith`) if the pointer may contain a value that is potentially obtained from a source relevant to that category (e.g., the result of a pointer arithmetic operation, the return value of an external call, a value of type `* const void` or `* mut void`) or if its value may flow into a sink relevant to that category (e.g., pointer arithmetic, or an argument to an external call, or a value that is cast to a `void` pointer).

Tables 4 to 6 contain the results of our analysis. Tables 4 and 5 show that 76.9% of raw pointer declarations, and 80.8% of raw pointer dereferences use pointers that are (sometimes indirectly) involved in multiple causes. The highest unique cause of raw pointer declarations and dereferences is the `Lifetime` category (9.5% and 10.0% respectively). However, the most prominent cause may depend on the program. For example, the highest contributing categories (in all 3 metrics) are `VoidPtr` in bzip2 which uses `void *` for polymorphism in order to share code between encoding and decoding stages, and `PtrArith` in TI which is a time series analysis library using and passing around dynamically allocated arrays. Finally, 71.4% of the functions use raw pointers for more than one reason, and 15.9% of these functions use pointers stemming from only `Lifetime`.

Table 4. Raw pointer declarations, grouped by category. ∃! and $\exists_{\geq_2}$ are explained in Section 2.2. Lifetime category contains only unique (∃!) causes by definition.

Benchmark	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	∃!	$\exists_{\geq_2}$	∃!	$\exists_{\geq_2}$	∃!	$\exists_{\geq_2}$	∃!	
libcsv	7	10	0	7	2	3	18	37
urlparser	0	70	0	70	4	70	5	79
RFK	0	0	1	0	0	0	1	2
genann	0	61	0	62	6	62	5	73
lil	1	314	60	316	10	317	50	438
json-c	13	227	1	227	9	211	41	297
bzip2	43	112	24	93	9	112	37	227
libzahl	9	324	114	322	3	319	7	457
TI	15	41	724	41	4	41	82	866
tinycc	18	1,100	16	1,094	23	1,084	192	1,352
optipng	12	1,016	121	987	48	1,013	210	1,407
libxml2	445	8,389	170	7,787	141	8,393	800	9,950
Total	563	11,664	1,231	11,006	259	11,625	1,448	15,185

Table 5. Raw pointer dereferences, grouped by category. ∃! and $\exists_{\geq_2}$ are explained in Section 2.2. Lifetime category contains only unique (∃!) causes by definition.

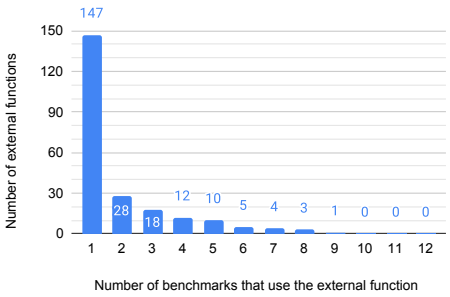
Benchmark	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	∃!	$\exists_{\geq_2}$	∃!	$\exists_{\geq_2}$	∃!	$\exists_{\geq_2}$	∃!	
libcsv	0	26	0	26	0	17	148	174
urlparser	0	2	0	2	0	2	58	60
RFK	0	0	24	0	0	0	0	24
genann	0	312	22	313	4	313	0	339
lil	0	895	127	897	8	897	636	1668
json-c	9	1639	39	1646	56	1433	93	1843
bzip2	1704	1192	173	627	11	1195	679	3764
libzahl	1	1220	1183	1220	22	1191	31	2457
TI	426	184	1237	184	0	184	0	1847
tinycc	28	4525	122	4522	9	4491	946	5632
optipng	5	5212	203	5043	36	5208	606	6062
libxml2	986	46536	303	42237	235	46543	4472	52546
Total	3159	61743	3433	56717	381	61474	7669	76416

2.3.2 ***External Function Calls.*** We break this investigation down into two questions: (1) How prevalent are calls to specific external functions? (2) Which external functions have the highest impact on safety? To answer these questions, we focus on the external functions and the internal functions that are, directly or transitively, made unsafe due to calls to those external functions. These internal functions may be unsafe for other reasons as well, but for this investigation we ignore other causes of unsafety.

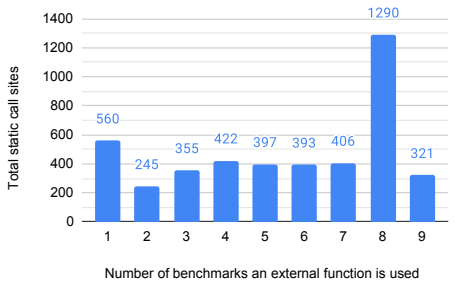
How prevalent are the external functions across benchmarks? It would be useful to know if there are a small set of external functions that appear across many benchmarks, making them an attractive target for replacement. There were 228 external functions used in total across all of our benchmarks. Figure 1a breaks down these external functions based on how many benchmarks use each one. We observe that 64% (147/228) of the external functions are unique to a particular

Table 6. Functions using raw pointers in a given category. $\exists!$ and $\exists_{\geq 2}$ are explained in Section 2.2. Lifetime category contains only unique ($\exists!$) causes by definition.

Benchmark	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	
libcsv	1	5	0	3	0	4	12	18
urlparser	0	5	0	5	0	5	2	7
robotfindskitten	0	0	0	0	0	0	2	2
genann	0	6	0	6	0	6	3	9
lil	1	62	9	61	0	62	6	78
json-c	6	50	0	50	1	47	20	77
bzip2	7	28	2	19	0	28	4	41
libzahl	0	79	9	79	0	79	2	90
tulipindicators	2	3	96	3	0	3	45	146
tinycc	3	76	5	75	0	65	35	119
optipng	4	175	12	171	5	177	62	260
libxml2	7	536	6	510	7	540	36	596
Total	31	1025	139	982	13	1016	229	1443



(a) Number of external functions based on how many benchmarks use them



(b) Number of calls to external functions based on how many benchmarks use them

Fig. 1. Counts of external function calls.

benchmark, and that no external function is used by more than 9 (out of 12) of our benchmarks. The external functions that occur in more than half of our benchmarks are:

- Used in 9 benchmarks: `strcmp`
- Used in 8 benchmarks: `strlen`, `printf`, `fprintf`
- Used in 7 benchmarks: `memcpy`, `fopen`, `fclose`, `exit`

Most of these functions deal with string manipulation or I/O. Figure 1b divides the external calls into bins based on how many benchmarks use them, then counts for each bin how many static call sites to a function in that bin appear across the benchmarks. For example, the column labeled '2' shows that there were 245 static call sites to an external function that appears in exactly two benchmarks. The total calls to functions used in exactly 8 benchmarks is much higher due to `fprintf`, which is called from 858 places across the benchmarks. We can see that the external functions listed above as common across 7 or more benchmarks account for almost half (46.0%) of all external function calls.

What are the external functions with the highest impact on unsafety? Another useful statistic for prioritizing external functions is their relative impact on unsafety. In order to measure this factor, we investigate:

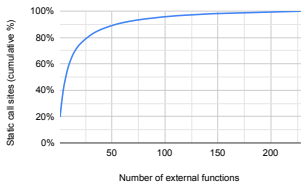
- The number of static call sites for each external function
- In how many functions an external function is called (directly or transitively)

Figure 2a shows an optimal ordering of external functions that maximizes cumulative static call sites. Overall, only seven external functions need to be replaced by safe alternatives to eliminate more than half (52%) of the external function calls. The most common functions in this ordering are similar to the most common functions reported above. The ten most commonly called functions we encounter in order are: `fprintf`, `strcmp`, `memset`, `printf`, `memcpy`, `strlen`, `snprintf`, `__assert_rtn`, `__ctype_toupper_loc`. Here, `__assert_rtn` is the C library function used in implementing the `assert` macro which can be replaced by Rust's safe `assert!` macro, and `__ctype_toupper_loc` is an implementation detail of the `toupper` function in C which has a safe counterpart in the Rust standard library.

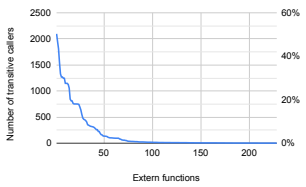
The other statistic we focus on is the external functions that make the highest number of functions unsafe (that is, the external functions that are called from the most functions, directly or transitively). Figure 2b shows how many transitive callers each external function has, both as absolute value and percentage. The ten extern functions that have most transitive callers in our benchmarks in order are: `memset`, `memcpy`, `__xmlRaiseError`, `strlen`, `snprintf`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_init`. Each of these functions contribute to the unsafety of 32.5–54.6% of the extern-calling functions. Here, `__xmlRaiseError` and the `pthreads`-related functions are used only by our largest benchmark, `libxml2`. `__xmlRaiseError` is an external function because of how libxml2 is linked: some features such as error reporting are linked from support modules that are compiled separately from the main program. This fact shows that an effort to make the whole benchmark project link in an idiomatic way for a Rust program can reduce pervasive external calls.

Some of the most-called external functions above are specific to a single benchmark. To assess the impact of external functions that are not specific to one benchmark, we applied the same analysis restricted to external functions used in more than one benchmark. Figure 2c shows how many transitive callers each external function used in more than one benchmark has. The 10 most called external functions with this restriction in order are: `memset`, `memcpy`, `strlen`, `snprintf`, `fprintf`, `memmove`, `__errno_location`, `memcmp`, `strchr`, `strcmp`. These functions contribute to the unsafety of 19.9–54.6% of the functions. Each function in this list is called in at least 4 benchmarks, except `__errno_location`, which is called in 3 benchmarks and it comes from accessing the `errno` variable in the C standard library. This list is similar to the previous list for the prevalence question in that it consists mainly of string manipulation, copying/initializing arrays in memory, and I/O.

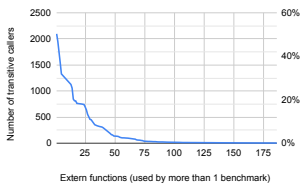
2.3.3 ***C-Style Unions.*** We manually inspected all C-style unions declared in our benchmarks. Most of these were defined by the C developers with accompanying tag data in order to manually implement a tagged union. In some benchmarks, the tag information was not stored with the union data but rather inferred from invariants that hold at a given program point. `libxml2` contains declarations for pthreads-related unions used in external calls; however, these unions are used only by pthreads functions and never read directly by the Rust program so they don't contribute to unsafety. Apart from these, none of the unions in our benchmarks are passed to or obtained from external functions, and we conjecture that they can be replaced with safe tagged unions (Rust `enums`) to reduce the use of C-style unions in the program. However, this transformation would



(a) Cumulative number of calls in program text for external functions, x-axis is ordered from the functions with most call sites to least



(b) Number of transitive callers each extern function has, ordered by most to least.



(c) Number of transitive callers each extern function used by more than one benchmark has, ordered by most to least.

Fig. 2. Impact of external function calls on unsafety.

yield highly un-idiomatic Rust code which would check the type of the union twice in the cases where there is an explicit tag that the C program checks.

2.3.4 ***Unsafe Casting.*** We inspected the calls to `mem::transmute` generated by `c2rust`. There are two uses of unsafe casting in the generated benchmarks: (1) converting 8-bit byte arrays to C character arrays (different from Rust strings) which corresponds to 71.7% (296 out of 413) instances of unsafe casting, and (2) converting between function pointers[7] and `void *` which corresponds to the remaining 28.3% (117) unsafe casts. The first option is safe on architectures using 8-bit unsigned characters (most modern architectures), and can be put behind a wrapper function.

2.4 Translated Rust versus Native Rust

Comparing ourselves to prior studies on the sources of unsafety in native Rust programs [Astrauskas et al. 2020; Qin et al. 2020], we asked ourselves: *What are the similarities and differences in the distributions of causes of unsafety between native Rust programs and Rust programs automatically translated from C?*

The closest work to ours in the domain of quantifying unsafety in Rust programs is Astrauskas et al. [2020]'s work on classifying how programmers use unsafe Rust. They classify Rust programmers' reasons for using `unsafe` and quantify the prevalence of each reason in the open source Rust ecosystem. However, our methodology differs significantly, so that comparing their results and our own is not straightforward. We consider root causes of unsafety in a function (directly and transitively) whereas Astrauskas et al. [2020] considers only the function body itself; they also use only coarse-grained unsafety categories. For example, their study lists "unsafe function calls" as the only provided reason for unsafety for 83.5% of the functions in their benchmarks, which as we have described above includes calls to unsafe internal functions, unsafe external functions, and unsafe compiler intrinsics, as well as unsafe casting and inline assembly. The differences in our methodologies partially stem from the differences between our goals and theirs:

- Astrauskas et al. [2020] attempts classify how Rust programmers use `unsafe`, so causes of unsafety in the function body is important to their study. Also, their classification of unsafe functions reflects their relation to the Rust ecosystem, e.g., unsafe functions from the standard library, the current program, and crates (packages in Rust ecosystem) providing Rust bindings to existing C/C++ code.

[7]Function pointers are represented in Rust as optional references rather than raw pointers, so casting them directly to and from raw pointers is unsafe.

- We are interested in classifying and quantifying the prevalence and impact of unsafe behavior in programs translated from C. In our case, the unsafe features are included because the translation mimics the original C program as faithfully as possible, not because Rust programmers consciously used the unsafe behavior. Also, our classification of unsafe functions reflects more fine-grained reasons for what causes the unsafety (e.g., memory allocation or unsafe casts).

Excluding their category "use of unsafe functions" (and the corresponding categories in our own classification), the list of reasons for unsafety given in their study, ordered by quantity, are in the same order as our own study: raw pointer dereference, access to mutable/external globals, unions, and inline assembly. However, the ratios of these sources of unsafety differ, possibly because of the differences in methodology. Also, we do not observe certain rare reasons (occurring in 0.1% of unsafe function bodies) of unsafety that they report in their Table 3 [Astrauskas et al. 2020], as none of these would be generated by `c2rust`.

2.5 Observations and Discussion

From Table 3, we can see that most functions are affected by multiple categories of unsafety: for each category, the number of functions uniquely affected by that category is 0–2% of the total number of functions affected by that category, with RawDeref being an outlier at 15%. Unions, inline assembly, and casts never appear by themselves at all. These numbers indicate that making translated Rust programs safer is a multi-faceted problem, in that fixing a single category of unsafety will not make a large impact on the number of unsafe functions. Only by fixing multiple categories can we hope to make a significant difference.

Because an effective method for making translated Rust programs safe needs to handle multiple categories of unsafety, an interesting question is how to prioritize which categories to handle. To answer this question, we graph the cumulative impact of fixing categories highest-to-lowest according to the following heuristic order of impact: *raw pointer dereference, memory allocation, extern calls, access to globals, unsafe casts, access to unions, inline assembly.* To assess the potential of solving these problems in this order we calculate the cumulative impact of how many unsafe functions become safe as each of these categories of unsafety are eliminated. Figure 3 shows the results of this calculation. We include both the results for all functions in all benchmarks, and the result for the three largest benchmarks in order to demonstrate the variability of the results. In this graph we include the functions unnecessarily marked unsafe. The results on the graph indicate that, in order to make more than half of the functions safe, we need to handle the four most common sources in our list. Also, the the graph (along with the tables) shows that impact of unsafe casts and C-style unions vary considerably depending on the benchmark program.

Another important point for translating C to *safer* Rust is the potential strategies for making the translated programs safer. Ultimately, unsafety stems from the fact that the compiler does not have enough information about a piece of code (e.g. the underlying types in the case of `void` pointers, or the code being executed in the case of external functions), and some of these features such as untagged unions, and void pointers are resorted to by programmers because of C's lack of language features. In all cases except external functions and internal assembly, the translator would have access to the code being executed, and it can use heuristics or other logics besides Rust's type system to derive the information to provide to the compiler. We propose a method in the next section with this underlying idea to derive the lifetime and borrowing information the Rust compiler needs to make raw pointers safe. We also focus on specifically raw pointers that fall in `Lifetime` category, as they do not have other causes of being raw pointers, and the techniques for making them safe would also be used for determining lifetimes of other categories of raw pointers.

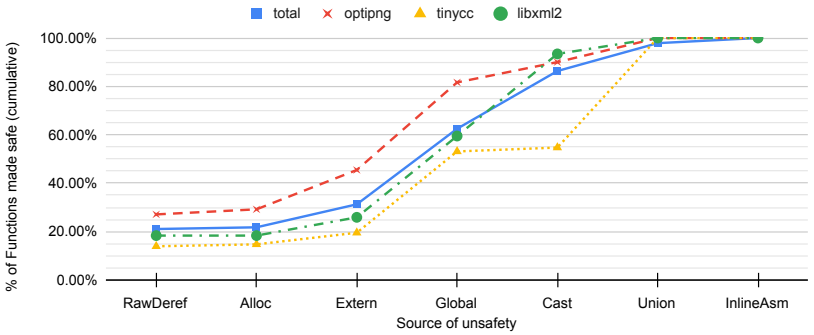


Fig. 3. Cumulative percentage of functions made safe by fixing the given unsafety category. The "Total" line shows this number for all functions across all benchmarks.

Finally, we need to make a distinction between the abstractions the two languages use. Technically safe Rust code translated directly from C and idiomatic Rust code can look completely dissimilar because of the abstractions Rust and the Rust standard library provide. For example, safe code translated from C may use `while` loops and indexing to go over arrays and other data structures whereas an idiomatic Rust program would use mutable and immutable iterators, and higher-order functions such as `map` for the same purpose purpose. Similarly, the C program may not have written in a way that is not amenable to direct translation due to the algorithms and data structures being used (such as graphs represented with complex and implicit ownership semantics). To solve both of these problems, the translation tools will need to operate at a higher level than translating the direct operational semantics of a program.

3 TURNING RAW POINTERS INTO SAFE REFERENCES

In this section we describe a first attempt to automatically translate a C program into a safer Rust program, building on top of the `c2rust` syntactic translation from C to completely unsafe Rust. Our study shows that, while addressing only a single category of unsafety reasons will not be sufficient to remove a majority of the unsafety in the translated program, the `Lifetime` category of raw pointers is a good place to start. Thus, our goal is to translate a subset of the raw pointers in the `Lifetime` category to safe references. We first start with a high-level description of our technique in this section; more details are provided in Section 4.

A raw pointer can be converted into a safe reference if, in the resulting rewritten program, the Rust compiler can prove that the reference guarantees a single owner and can also derive the appropriate lifetime for the object being referred to. One possible approach would be to implement a static analysis for either the original C program or the translated unsafe Rust program to compute this information; however, the drawbacks of such an approach are: (1) designing and implementing an efficient, useful analysis that can reason about aliasing and lifetime information in conformance with Rust's sophisticated type system is highly non-trivial; and (2) even if the implemented analysis can prove safety, that doesn't matter unless the Rust compiler can *also* prove safety, i.e., the analysis must be tuned to be no more precise than the Rust compiler.

Our key insight is that we can piggy-back on top of the Rust compiler and allow it to derive the information we need to infer which `Lifetime` raw pointers can be made safe. To do so, we first optimistically rewrite the unsafe Rust program to convert all of the relevant raw pointers into safe references, making optimistic assumptions about mutability, aliasing, and lifetimes. This optimistic version is very unlikely to compile—but the errors that the Rust compiler derives while attempting

to compile it allow us to refine our initial optimistic program into a more realistic version. By iterating this process in a loop, we essentially use the Rust compiler as an oracle to continually refine the program until it passes the compiler. For this first attempt we do not try to introduce any additional memory management mechanisms (e.g., reference counting) that might allow more raw pointers to become safe, focusing purely on converting raw pointers into safe references with the same memory representation and performance characteristics; future work will investigate these other possibilities.

During our translation, we assume that any pointers passed to an API are valid pointers (`null` or a valid reference to an object) if the program dereferences them, because dereferencing an invalid pointer would result in undefined behavior in both C and Rust. Therefore, these raw pointers could be converted to safe references without changing the *defined* program behavior, if their use does not invalidate Rust's borrow checker rules.

Our technique consists of three stages after the `c2rust` translation of the original C program:

(1) Connect the definitions and uses of types and functions across modules, and remove unnecessary unsafety and mutability markers. (Section 3.1)

(2) Determine initial lifetimes to convert unsafe raw pointers into safe references. (Section 3.2)

(3) Iteratively rewrite the program to resolve lifetime inference and borrow checking errors. (Section 3.3)

We will explain our technique in relation to an example C program shown in Figure 4a, which implements a binary search tree. Figure 4b shows the result of running `c2rust` on the C program. We will step through each stage of our technique in the remainder of this section, demonstrating on the given example.

3.1 Connecting functions and data structures across modules

The original C program may consist of multiple compilation units (e.g., Figure 4a has two: `bst.c` and `main.c`). `c2rust` translates each compilation unit separately into its own Rust module (e.g., Figure 4b has `bst.rs` and `main.rs`). However, unlike C, all Rust modules in a program are compiled together in the *same* compilation unit. Because `c2rust` translates each C compilation unit separately, the translated modules contain (1) duplicate data structure declarations from shared header files; and (2) functions declared as `extern` because they are defined in a different module, even though the definitions are actually available during compilation. In Figure 4b, note that `main.rs` contains a duplicate declaration of `node_t` and declares both `find` and `insert` as extern functions. All calls to declared extern functions must be marked `unsafe`, regardless of the fact that the functions are not truly extern. The result is that, even if we manage to make `find` and `insert` safe in the `bst.rs` module, `main_0` must remain unsafe because it contains calls to those functions and they were declared extern in the `main.rs` module.

The immediate solution is to remove the extern declarations and replace them with imports from the modules in which those functions are defined. However, doing so can cause a type error if the functions use a data structure that has been duplicated across modules. Rust's type system is nominal, and these duplicated definitions are treated as separate types. In Figure 4b the type `bst::node_t` and the type `main::node_t` are two different types; because the formerly extern functions are now imported and use the duplicated type, there is now a type error in the example program. In order to fix this issue, we need to detect and deduplicate these data structure declarations. After this step, we remove unnecessary `mut` markers and `unsafe` markers. For our example, the only unnecessary `mut` markers are in the arguments of `find` and `insert`. All the `unsafe`

```c
// bst.h: BST node definition
typedef struct node_t {
  node_t* left;
  node_t* right;
  int value;
};

node_t* find(int value, node_t* node);
void insert(int value, node_t* node);

// bst.c: BST implementation
#include "bst.h"

node_t* find(int value, node_t* node) {
  if (value < node->value && node->left) {
    return find(value, node->left);
  } else if (value > node->value && node->right)
      {
    return find(value, node->right);
  } else if (value == node->value) {
    return node;
  }
  return NULL;
}

void insert(int value, *node_t node) {
  // Implementation omitted for brevity.
}

// main.c: program entry point
#include "bst.h"

int main() {
  node_t* tree = malloc(sizeof(node_t));
  tree->value = malloc(sizeof(int));
  *(tree->value) = 3;
  insert(1, tree);
  insert(2, tree);
  *(find(3, tree)->value) = 4;
  return 0;
}
```

(a) A C program implementing a binary search tree. We omit the implementation of insert for brevity.

```rust
// bst.rs
use std::os::raw::c_int;

#[derive(Copy,Clone)]
pub struct node_t {
  pub left: *mut node_t,
  pub right: *mut node_t,
  pub value: c_int,
}

pub unsafe fn find(mut value: c_int, mut node:
    *mut node_t) -> *mut node_t {
  /* ... */
}

pub unsafe fn insert(mut value: c_int, mut node:
    *mut node_t) { /* ... */ }

// main.rs
use std::os::raw::c_int;
extern "C" {
  fn find(mut value: c_int, mut node:
      *mut node_t) -> *mut node_t;
  fn insert(mut value: c_int, mut node:
      *mut node_t);
}

// duplicate definition of node_t
#[derive(Copy,Clone)]
pub struct node_t {
  pub left: *mut node_t,
  pub right: *mut node_t,
  pub value: c_int,
}

pub unsafe fn main_0() -> int { /* ... */ }
```

(b) The Rust program produced from Figure 4a. Function bodies, `main`, and `main_0` are omitted for brevity, as are compiler directives for C compatibility (e.g. for disabling name mangling, ensuring C ABI, and structure field alignment).

Fig. 4. An example a of C program translated to Rust by `c2rust`.

markers in the example code are still necessary due to raw pointer dereferences. Figure 5 shows our example after this process.

3.2 Initial optimistic rewrite

The next stage is to rewrite the program into a version with no `unsafe` annotations due to `Lifetime` raw pointers (`unsafe` annotations due to other categories of unsafety will remain). Henceforth we will just refer to "raw pointers"; this term should be taken as `Lifetime` raw pointers. The rewriting process is optimistic in the sense that it will likely result in a non-compilable program. The first step of this stage is to rewrite raw pointer declarations (e.g., data structure fields and function parameters) into reference declarations. Specifically, we convert the raw pointers into optional references in order to account for null pointer values: `Option<&T>`, `Option<&mut T>` and `Option<Box<T>>` represent immutably borrowed, mutably borrowed, and owner pointers, respectively. We assume

```rust
// bst.rs
use std::os::raw::c_int;

#[derive(Copy,Clone)]
pub struct node_t {
  pub left: *mut node_t,
  pub right: *mut node_t,
  pub value: c_int,
}

pub unsafe fn find(value: c_int, node: *mut node_t) -> *mut node_t {
  if value < (*node).value && !(*node).left.is_null() {
    return find(value, (*node).left)
  } else {
    if value > (*node).value && !(*node).right.is_null() {
      return find(value, (*node).right)
    } else if value == (*node).value {
      return node
    }
  }
  return 0 as *mut node_t;
}
pub unsafe fn insert(value: c_int, node: *mut node_t) { /*...*/ }

// main.rs
use std::os::raw::c_int;
use bst::{node_t, insert, find};

pub unsafe fn main_0() -> int {
    let mut tree = malloc(::std::mem::size_of::<node_t>()) as *mut node_t;
    (*tree).value = malloc(::std::mem::size_of::<c_int>()) as *mut c_int;
    *(*tree).value = 3;
    insert(1, tree);
    insert(2, tree);
    *(*find(3, tree)).value = 4;
    return 0;
}
```

Fig. 5. The Rust program from Figure 4b after deduplicating struct definitions and converting extern functions to imports. The unnecessary mutability annotations have been removed from the function arguments.

for this stage that all declarations are borrowed; the third, iterative stage may later convert them into owners instead.

When declaring a reference in function signatures or data type definitions, we must provide its lifetime information. This information includes the lifetime of the reference itself and also the information for any referenced types that are themselves parameterized by lifetime. Our goal for this stage is to generate lifetime information that minimally constrains the declarations, in order to start with the most optimistic lifetime assumptions.

For each raw pointer data structure field we provide a lifetime based on its type, using a different lifetime variable for each type.[8] We also fill in lifetime type parameters, using the same lifetime variables for all instances of the same type. Mutably borrowed references are not cloneable (i.e., trivially copyable), so we remove the `#[derive(Copy,Clone)]` annotation from any affected data structures. For our example program, the end result of rewriting the `node_t` data structure is:

```rust
pub struct node_t<'a1, 'a2> {
    pub left: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub right: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub value: Option<&'a2 mut c_int>,
}
```

[8] We could also give each field a unique lifetime, but this type-based heuristic works well empirically and makes it easy to handle recursive type declarations.

```
pub fn borrow<'b, 'a: 'b, T>(p: &'b Option<&'a mut T>) -> Option<&'b T> {
    p.as_ref().map(|x| &**x)
}
pub fn borrow_mut<'b, 'a: 'b, T>(p: &'b mut Option<&'a mut T>)
  -> Option<&'b mut T> {
    p.as_mut().map(|x| &mut **x)
}
```

Fig. 6. Helper functions which assist in rewriting pointers to references. They allow borrowing an optional reference for a *shorter* lifetime, where `'a` is the original object's lifetime and `'b` is the borrowed object's lifetime.

Once the data structures are rewritten, we rewrite the function signatures in accordance with the new declarations, again making all raw pointers into borrows. Unlike data structure fields, for function signatures we use a unique lifetime for each parameter. For our example, the rewritten function signatures for `find` and `insert` are:

```
fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>) ->
    Option<&'a4 mut node_t<'a5, 'a6>>;
fn insert<'a1, 'a2, 'a3>(value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>);
```

The signature of `main_0` does not change, since it does not involve any pointers. Next we rewrite function bodies, which entails four types of rewrites:

(1) We rewrite any call to `malloc` that allocates a single object (as opposed to an array) into a call to `Box::new`, a standard Rust function for safe heap allocation. We determine which `malloc` calls to rewrite by checking for calls that are translated from `malloc(sizeof(T))` in the C program.

(2) We delete any call to `free` if we can replace all pointers that are freed at that call site with safe references. If we cannot replace all such pointers, then we need to keep the call to `free` so we roll back any pointers reaching this `free` that were previously rewritten.

(3) We rewrite any equality comparisons between references, which by default are value equality checks in Rust (i.e., checking equality of the objects being referenced), into a reference equality check (i.e., checking whether two references refer to the same object). This rewrite preserves the intended semantics of the original program.

(4) Dereferences must be rewritten to unwrap the optional part of the reference (recall that we replaced the raw pointer with an *optional* reference). Unwrapping the option consumes the original `Option` object because `Option<T>`, unlike raw pointers, is not automatically copyable. Therefore, we do the following to avoid consuming the original object in the contexts that it is not assigned to or deliberately consumed:

- When using an immutable reference, we clone it so the original object is not destroyed;
- When using a mutable reference, we make a mutable or immutable borrow depending on the context it is used in. We describe how we create these borrows below.

To help with re-borrowing mutable references, we use the helper functions `borrow` and `borrow_mut` defined in Figure 6. For each pointer `p` in the original program that we converted to a mutable reference, we perform the following rewrites:

- If `p` is passed to a mutable context (a context requiring a `&mut T`), we rewrite `p` to `borrow_mut(p)`.
- if `p` is passed to an immutable context (a context requiring a `&T`), we rewrite `p` to `borrow(p)`.
- if `p` is dereferenced, we rewrite `*p` as `**p.as_mut().unwrap()` to get a mutable reference and immediately dereference it. If it is dereferenced in an immutable context, we use `as_ref` instead of `as_mut`. Note that `unwrap`, `as_mut`, and `as_ref` all come from the Rust standard library.

```rust
// bst.rs
use std::os::raw::c_int;

pub struct node_t<'a1, 'a2> {
    pub left: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub right: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub value: Option<&'a2 mut c_int>,
}
impl<'a1, 'a2> std::default::Default for node_t<'a1, 'a2> {
  // ...
}
pub fn insert<'a1, 'a2, 'a3>(mut value: c_int,
                             mut node: Option<&'a1 mut node_t<'a2, 'a3>>) {
    // ...
}
pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>)
 -> Option<&'a4 mut node_t<'a5, 'a6>> {
    if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).left.
      is_none() {
        return find(value, borrow_mut(&mut (*node.unwrap()).left))
    } else {
        if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).right.
      is_none() {
            return find(value, borrow_mut(&mut (*node.unwrap()).right))
        } else { if value == **(**node.as_mut().unwrap()).value.as_mut().unwrap() { return node } }
    }
    return None;
}

// main.rs
use std::os::raw::c_int;
use bst::{node_t, insert, find};

pub fn main_0() -> int {
    let mut tree = Some(Box::new(node_t::default()).as_mut());
    **(**tree.as_mut().unwrap()).value.as_mut().unwrap() = 3;
    insert(1, borrow_mut(&mut tree));
    insert(2, borrow_mut(&mut tree));
    **(**find(3, borrow_mut(&mut tree)).as_mut().unwrap()).value.as_mut().unwrap() = 4;
    return 0;
}
```

Fig. 7. The Rust program from Figure 5 after converting raw pointers into references.

We rewrite null pointers into `None`, i.e., the `Option` value that does not contain anything. We similarly rewrite the null pointer check `p.is_null()` into `p.is_none()`. Figure 7 shows our example program after all of these transformations.

3.3 Iteratively rewriting the program until it compiles

The initial, optimistic rewrite may have resulted in a non-compilable program, i.e., one for which the Rust compiler cannot prove safety. The last stage of our technique iteratively attempts to compile the program with the Rust compiler; for each failed attempt we take information from the compiler errors to selectively rewrite our optimistic changes until we reach a version that compiles. These rewrites in some cases provide the compiler with more refined lifetime information or modify reference types, while in other cases we are forced to walk back on the changes and leave some raw pointers as unsafe. When a version of the program fails to compile, we track the following information:

- Any additional lifetime constraints the compiler reports. For example, when compiling the program in Figure 7 the compiler reports that for `find` there is an additional constraint `'a1`

: 'a2, meaning 'a1 needs to outlive 'a2. For the next iteration we rewrite the program to explicitly include this constraint and any additional constraints learned from similar errors.

- The references involved when a reference outlives an object. If the original object is on the heap, we promote the reference to an owned object on the heap and move the object instead of borrowing it, i.e., converting from Option<&T> to Option<Box<T>>. If the original object was on the stack, then we demote these references to raw pointers.
- If any rewritten malloc and free calls were involved in the failure. Rewritten calls can fail to compile when the original C program uses magic numbers or a custom allocation pattern. In subsequent iterations we do not attempt to rewrite any values that come from these particular calls to malloc, or that flow into these particular calls to free.
- The references involved in either use-after-move errors or multiple mutable borrow errors. We rewrite these references back to raw pointers.

When we demote a reference back to a raw pointer, we need to make all other references that interact with that demoted reference into raw pointers as well. We use the taint analysis from Section 2.3.1 to propagate the required information about any references we decide to convert back to raw pointers because of borrow errors. Similarly, if we decide to make a reference owned, all the values that flow into it must also be owned. We propagate these facts by performing a subset-based version of the taint analysis we used in Section 2.3.1 and marking the references promoted to owned references as sinks.

We demonstrate these steps on the example program in Figure 7. For this example we do not encounter issues involving the last two cases above.

The first compilation attempt fails with a compiler error stating that the following lifetime constraints are not satisfied: 'a1 : 'a4, 'a5 : 'a2, and 'a6 : 'a3. All of these constraints come from the return node; statement on line 23, and they are all rooted in the fact that the reference find returns cannot outlive its argument. Specifically, 'a1 : 'a4 comes directly from the references, and the other two constraints come from the fact that the data structures are covariant on their lifetime arguments and the functions are contravariant on their lifetime arguments. To resolve the errors we add these constraints to the signature of find and continue iterating.

The second compilation attempt also fails, this time with a compiler error stating that recursive calls to find require the additional constraints 'a2 : 'a5 and 'a3 : 'a6. We add these constraints as well, and continue iterating.

The third compilation attempt fails again, with a compiler error stating that we return a value that cannot outlive borrowing node in lines 19 and 22. To resolve the error we rewrite the borrows in these dereferences **node.as_mut().unwrap() as *node.unwrap(), ultimately consuming the reference node. This heuristic works for many of the cases in our benchmarks, but it might create use-after-move errors later on, in which case we would walk the rewrites back and make the node parameter of find a raw pointer again. In addition we get another lifetime error indicating that the variable tree in main function outlives the object it references (line 33), the temporary boxed object. To fix this error we we convert tree to be an owned object (Option<Box<node_t>>).[9] Now that tree is an owned reference, we rewrite the places it is borrowed as tree.as_mut().map(|b| b.as_mut()) to get a mutable reference inside the Option without consuming tree. We need to propagate the fact that tree is now an owned reference to all the values that flow into tree. After using our taint analysis

[9]We could potentially make it a Box<node_t> without the Option part because it is never assigned to a value containing None, however we apply the same strategy independent of the position (including struct fields) and we need the optional types when creating default values for struct fields.

to propagate this fact, we discover that the box at line 33 should be an owning reference, so we make that expression own the allocated object by removing the call to `as_mut()` on that line.

After these rewrites, the program compiles and all raw pointers have been converted into safe references. Note that we omitted the implementation of `insert` in this example to keep the number of steps shorter. Figure 8 shows the final fixed program.

```
1  // bst.rs
2  use std::os::raw::c_int;
3  // BST node
4  pub struct node_t<'a1, 'a2> {
5      pub left: Option<&'a1 mut node_t<'a1, 'a2>>,
6      pub right: Option<&'a1 mut node_t<'a1, 'a2>>,
7      pub value: Option<&'a2 mut c_int>,
8  }
9  impl<'a1, 'a2> std::default::Default for node_t<'a1, 'a2> {
10   // ...
11 }
12 pub fn insert<'a1, 'a2, 'a3>(mut value: c_int,
13                               mut node: Option<&'a1 mut node_t<'a2, 'a3>>) {
14     // ...
15 }
16 pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>)
17  -> Option<&'a4 mut node_t<'a5, 'a6>>
18 where 'a1: 'a4, 'a5: 'a2, 'a6: 'a3, 'a3: 'a6, 'a2: 'a5
19 {
20     if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).left.
       is_none() {
21         return find(value, borrow_mut(&mut (*node.unwrap()).left))
22     } else {
23         if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).right.
       is_none() {
24             return find(value, borrow_mut(&mut (*node.unwrap()).right))
25         } else { if value == **(**node.as_mut().unwrap()).value.as_mut().unwrap() { return node } }
26     }
27     return None;
28 }
29
30 // main.rs
31 use std::os::raw::c_int;
32 use bst::{node_t, insert, find};
33
34 pub fn main_0() -> int {
35     // Using Box to avoid malloc clutter
36     let mut tree = Some(Box::new(crate::node_t::default()));
37     **(**tree.as_mut().unwrap()).value.as_mut().unwrap() = 3;
38     // insert 2 nodes
39     insert(1, tree.as_mut().map(|b| b.as_mut()));
40     insert(2, tree.as_mut().map(|b| b.as_mut()));
41     // change the value of node containing 3 to 4
42     **(**find(3, tree.as_mut().map(|b| b.as_mut())).as_mut().unwrap()).value.as_mut().unwrap() = 4;
43     return 0;
44 }
```

Fig. 8. The safe Rust program with no raw pointers after applying all steps of our technique.

4 METHOD IN DETAIL

This section provides details of our method, which was previously outlined in Section 3.

4.1 Minimizing the number of unsafety and mutability markers

As a first step, we want to minimize the number of unnecessary `unsafe` and mutability annotations in the program. Fewer mutability annotations generally permits more safe programs, as mutable references are inherently more restricted.

4.1.1 Deduplicating function and data structure declarations. As explained in Section 3.1, differences in the granularity of translation units between C and Rust cause functions to be unnecessarily externally-linked and data definitions to be duplicated. We resolve these issues with the following transformations:

- C does not have name mangling, so all external functions of the same name refer to the same functions in a linked program. This allows us to rewrite external function declarations with the same name into Rust import statements.
- In most C programs, the struct definitions with the same name and same field names refer to the same type across multiple translation units. This allows us to aggregate type definitions according to their names and fields, arbitrarily choose one as the canonical version, then import the canonical version from other modules.
- Similar to structs, we rewrite other external type declarations used in external function APIs as imports to types with the same name.

After these transformations, we can safely remove the `unsafe` annotation from functions lacking unsafe behavior.

4.1.2 Removing unnecessary mutability markers. We perform a simple global flow-insensitive taint analysis to check whether a place (i.e., a variable, return value, field access, or dereference) is ever used in a mutable context (i.e., assignment, passed to an external function as a mutable pointer, or mutably borrowed). We mark all values flowing into these places as mutable. We assume that the types of function pointers and the signatures of externally-linked functions are accurrate when performing this analysis. We then remove any mutability annotations this analysis did not derive from the program.

4.2 Creating the most strict program with initial lifetimes

Next, we try to convert raw pointers into safe references. In order to do this, we need to assign lifetimes to the raw pointers, and define the relationships between the assigned lifetimes to tell the compiler which values are supposed to outlive which other values. We need to derive this information only for the lifetimes that occur in struct fields (Section 4.2.2) and function signatures (Section 4.2.3), as the type checker can infer the lifetimes in function bodies from these specifications. Because our representation of safe references has a different semantics than raw pointers (Section 4.2.1), we need to modify function bodies to get ultimately the same semantics (Section 4.2.4). Similarly, calls to `malloc` need to be converted to safe alternatives.

4.2.1 Different ways to represent a pointer. For our purposes, we identify three safe ways to hold a reference to an object in Rust:

- **Owned pointers (`Box<T>`)** represent a boxed `T` allocated on the heap. `T` is owned by this pointer, and will be deallocated when the pointer goes out of scope.
- **Immutably borrowed references (`&'a T`)** represent read-only references to `T` that can be freely copied. No copy may outlive the lifetime $'a$, which itself cannot outlive `T`.
- **Mutably borrowed references (`&'a mut T`)** represent mutable references to `T`, which may not be freely copied.

All three always refer to valid, non-null objects. However, since raw pointers may be null, we wrap each of these alternatives in `Option`, which permits the null-like value `None`. As such, we use `Option<&T>`, `Option<&mut T>`, and `Option<Box<T>>` for our representation. Rust guarantees that this `Option`-based representation has the same underlying memory representation as raw pointers, thanks to "null pointer optimization" [The Rust developers [n.d.]a].

Table 7. Rewriting a pointer-producing expression p to not consume the pointer. The borrow and borrow_mut functions are defined in Figure 6.

Pointer type	Immutable context	Mutable context
`Option<&T>`	`p.clone()`	Pointer use error
`Option<&mut T>`	`borrow(&p)`	`borrow_mut(&mut p)`
`Option<Box<T>>`	`p.as_ref().map(|b| b.as_ref())`	`p.as_mut().map(|b| b.as_mut())`

Borrowing and dereferencing optional references. Using optional references (e.g. `Option<&T>`) is less ergonomic than using a raw pointer (e.g. `* const T`) or an unwrapped reference (`&T`) because of copying semantics. Raw pointers do not have any aliasing requirements, so they are trivially copyable, as long as the pointer itself is unmodified (not the pointed value). Immutable references are similarly copyable. However, `Option<&T>` is not copyable, so it requires explicit cloning. Similarly, `Option<&mut T>`, and `Option<Box<T>>` require explicit borrowing when needed, even though `&mut T` and `Box<T>` will implicitly be borrowed when needed. When we have a pointer valued expression `p` used in a function body, we rewrite `p` according to Table 7. Note that we cannot borrow the value inside an immutable pointer in a mutable context; such a case is a violation of Rust's type system, and we would instead backtrack and keep the relevant pointer values as raw pointers.

When we need to dereference a pointer, we first borrow it with a mutability depending on the context, then call `Option::unwrap()` to get the borrowed reference inside. Calling `Option::unwrap()` is guaranteed to crash the program if the optional value contains a `None`. However, this case exactly corresponds to dereferencing a `null` pointer in the original program, which is undefined behavior in C. To illustrate dereferencing on an example, suppose we have a mutable pointer expression `p`, which we then dereference in an immutable context (`*p`). We would rewrite this to `*borrow(&p).unwrap()`, where `borrow` is defined in Figure 6.

Reference equality vs. value equality. Raw pointers and references have different equality semantics, where raw pointers use reference equality (as with pointers in C), and references use value equality of the underlying value being pointed to. To preserve the semantics of the original program, we convert equality checks on two references to call to a function that performs a reference equality check.

Null pointers. We rewrite null pointer checks (calls to `is_null()`) as calls to `is_none()`. `is_none` itself immutably borrows the `Option`, so we do not need to clone or borrow it in this case. In places where the original program has a null pointer constant, we rewrite it as `None`.

4.2.2 Rewriting struct definitions: field lifetimes and default values. When generating an initial set of lifetimes for a data structure definition, our goal is to make the data structure be as unconstrained as possible. As such, we initially start with all references being mutably borrowed instead of owned. We may convert them on a per-field basis in when fixing the errors from Rust's borrow checker (Section 4.3). Each field is generally given a distinct lifetime. However, we observe that fields of the same type in the same struct generally come from the same source, and therefore we heuristically give these the same lifetime.

Another place where we need to introduce lifetime variables in a struct body is the lifetime parameters of other structs in the definition. For example, consider these two structs:

```
1  struct Foo {
2    value: * const i32, // 32-bit signed integer
3  };
4
5  struct Bar {
```

```
6     foo: * const Foo,
7   };
```

When rewriting the struct `Foo`, we introduce a lifetime parameter for it, we also need an extra lifetime parameter in `Bar` to pass to `Foo`:

```
1   struct Foo<'a> {
2     value: & 'a i32,
3   };
4
5   struct Bar<'a, 'b> {
6     foo: & 'b Foo<'a>,
7   };
```

A naive approach to handle this case is to just add extra lifetime parameters for each struct in the definition that takes lifetime parameters. However, this would not work for recursive data types such as `node_t` from Section 3. We propose a general algorithm to generate lifetimes for arbitrary mutually recursive definitions with the following constraints on the generated lifetimes:

- All fields of the same type (or more generally, all fields of the types involved in the same mutually recursive type definition) in a definition share the same lifetime variable.
- All instances of the same struct in a definition share the same lifetime parameters. For example, in the definition of `node_t`, both instances of `node_t` for the left and the right subtree use the same lifetime parameters `'a1` and `'a2`.
- All references in a nested pointer are assigned the same lifetime. For example, `* const * const T` is converted to `Option<&'a Option<&'a T>>`.

Our algorithm is shown in Figure 9. The first step we take is building a points-to graph of all structs in the program and to label the edges with fields. A struct `Foo` points to another struct `Bar` if it has a (possibly nested) pointer field that points to a `Foo` object. In our algorithm, once we compute the strongly-connected components (SCCs), Each SCC corresponds to a set of mutually recursive struct definitions. After building the SCC points-to graph, we aggregate the labels from the original graph, and also add the source nodes for each label. For each struct definition, we collect the lifetime names on all edges reachable from that struct's SCC to determine its lifetime parameters. When rewriting a field f of a struct S to a borrowing reference, we use the assigned lifetime name from the edge that contains the label $S.f$.

Build a points-to graph of all structs in the program.
Label all edges in this graph with their fields.
Compute the SCCs of this graph.
Build the points-to graph between SCCs.

for all edge $SCC_1 \rightarrow SCC_2$ in the SCC graph **do**
  Label the edge with $(S_1.f_1, S_2.f_2, \ldots)$ where $S_1, S_2, \ldots$ are structs in SCC_1 and each corresponding field $f_1, f_2, \ldots$ points to a struct in SCC_2.
end for
Assign a unique lifetime variable to each edge in the SCC graph that correspond to a borrowing reference.

Fig. 9. Our algorithm for computing lifetime parameters of structs.

When allocating a struct on the heap (i.e., when we rewrite a `malloc` call), Rust requires initializing it with a default value to prevent reading from uninitialized memory. All fields (rather than the data they point to) in the structs we rewrite fall into one of these categories: raw pointers, optional references, primitive types, or other structs. We implement the `Default` trait for all of our structs, and the `Default` trait is implemented for `Option` and the primitive types so we use `Default::default` to generate the default values for the fields of these types. We initialize all raw pointers inside a struct (that is all the fields that are not converted to safe references) with null pointers.

We do not rewrite a struct's definition in the following cases:

- The struct is contained in (directly or through pointers) a type that is part of an external API. In this case, we cannot have lifetime guarantees because the external API may hold references to the value, keep a copy of it, or be responsible for its allocation/deallocation.
- The struct contains (not points to, but immediately contains) a (C-style) union. In this case, we cannot generate a default value for the struct, because there are not any well-defined default values for unions. One may opt into picking one of the variants and generating the default value for that field. In general, we keep unions out of the scope of our method, and this is an orthogonal issue.

In both cases, we mark the pointer-typed fields in the struct as raw pointers. We cover how this marking works in Section 4.3.1.

We do not rewrite unions, as they are out of the scope of our method; getting a value from a union is unsafe behavior, and it may allow forging invalid references. The Rust programs translated from C do not use Rust enums (sum types), and the C-style enums are just integers which do not contain any references, so the only flavor of algebraic data type we need to handle is structs. Our technique in this section can be extended to enums by considering the points-to edges from all possible variants.

4.2.3 Function signatures. After determining the lifetimes in struct fields, we rewrite function signatures. For each parameter in a function, if that parameter is not tainted by any of the cases of raw pointers we do not handle, we initially convert it to a borrowing reference. We assign a unique place for each lifetime in a function signature to make the function signature as generic as possible. In later stages, we add lifetime constraints between these lifetimes or remove some of them as some of the borrowing references are converted to raw pointers or owning references. Assigning a unique lifetime variable in each place may yield unwieldy function signatures (such as the signature of `find` in Figure 7). As we discover relationships between the lifetimes in the function signature, we could unify them for readability. For example, we can unify `'a2` and `'a5` in `find` because we generate the constraints `'a2 : 'a5` and `'a5 : 'a2`, proving their equality. This change does not affect the correctness of our result and we do not implement it in our prototype.

4.2.4 Function bodies. Inside function bodies, we recursively visit all expressions, maintain a record of whether the expression is used in a mutable or immutable context, and rewrite all uses of pointers according to Section 4.2.1. We rewrite calls to `malloc` and `calloc` that allocate a single object (e.g. `malloc(size_of<T>()) as * mut T`) using `Box::new`

We handle the following patterns of calling `malloc` and `calloc`, including their `* mut T` variants:

- `malloc(size_of<T>()) as * const T`
- `calloc(1, size_of<T>()) as * const T`
- `calloc(size_of<T>(), 1) as * const T`

These patterns cover all allocations of single objects in our benchmarks. Other patterns of allocations or custom allocation functions can be added as pattern matches to our tool.

We rewrite a call to `malloc(size_of<T>()) as * const T` as `Some(Box::new(T::default()).as_ref())` to produce an immutably borrowing reference. These references may become owning references if there are any lifetime errors involving them, in which case we remove the call to `as_ref`. If the result of `malloc` is cast into a mutable pointer, we use `as_mut` instead.

We remove calls to `free`, if all calls to `malloc` that flow into these `free` calls are rewritten. If any `malloc` that flows to a call to `free` cannot be rewritten, we undo rewriting all `malloc` calls that flow into that call to `free`.

4.3 Continually fixing lifetime errors

After creating the program with initial lifetimes, we incrementally modify the program using the following procedure:

(1) Generate the program corresponding to whatever the last set of changes to make was. Initially, this is an empty set.
(2) Run the Rust compiler on the program, and collect the lifetime errors. Then, for all errors:
  (a) Record the changes to fix each error.
  (b) If the changes involve promoting a location to an owned reference, mark all values flowing into that value also as owned.
  (c) If the changes involve promoting a location to a raw pointer, mark all values flowing from and to that location as a raw pointer (i.e., remove them from the scope of our method in the following iterations).
(3) If there are no errors, we are done. Otherwise, go to step 1.

In the following sections, we describe how we represent the sets of changes, and our analyses and heuristics for each type of error.

4.3.1 Configurations. There are two types of changes we perform on the original program:

(1) Converting raw pointers into owned or borrowed references, and
(2) Adding lifetime constraints.

We represent our sets of changes as *configurations*. A configuration consists of a mapping from program locations to the type of pointers they are converted to, along with corresponding lifetime constraints. We represent a poset of configurations using the definition in Figure 10. A configuration maps program locations (i.e., variables, parameters, return values, struct fields, and dereferences of any of these in case of nested pointers) to the kinds of pointers they should have. Each location also maps to the upper bound of each lifetime variable. A *qualified lifetime variable* is a lifetime variable qualified with which function signature or type definition it is used in; the second mapping maps qualified lifetime variables to the sets of variables in the same scope (function signature or type definition) that are upper bounds of it. As iterations progress, we move from smaller to larger configurations, where the largest configuration maps each reference to a raw pointer. Because configurations form a poset, and because there are finitely many locations and lifetime variables, this guarantees termination. Moreover, it guarantees we will terminate with a program containing the minimal number of raw pointers.

$$\begin{aligned}
\text{Configuration} &= (\text{Location} \to \text{PointerKind}) \times (\text{QualLifetimeVar} \to \mathscr{P}(\text{LifetimeVar})) \\
\text{QualLifetimeVar} &= (\text{Function} \cup \text{TypeName}) \times \text{LifetimeVar} \\
\text{PointerKind} &= \{\text{borrowed}, \text{owned}, \text{raw}\} \text{ where borrowed} \sqsubset \text{owned} \sqsubset \text{raw}
\end{aligned}$$

Fig. 10. The configurations of our program rewriting method.

4.3.2 Fixing lifetime errors we get from the Rust compiler. The Rust type and borrow checkers work in two stages, and they return the following lifetime errors:

(1) The type checker/region inferer assigns appropriate lifetimes (a.k.a. regions) to each value in the program. If it cannot assign lifetimes because it cannot match them according to the type signature, it gives a type error with the mismatching lifetimes.
(2) If the region inferer can assign the appropriate lifetimes, then the borrow checker checks if there are any forbidden uses of objects according to the lifetimes. The errors that stem from the borrow checker are:
   (a) use after move
   (b) moving a value while it is borrowed by a value that is still live
   (c) having multiple references that are alive and borrowing the same object, and at least one of them is mutable
   (d) the borrowed value does not live long enough
   (e) a local variable is returned by reference (i.e. borrowed)

In all of these cases, we use the data used by the compiler's diagnostics engine, along with the result of the type checker, to get the relevant lifetimes and expressions. Some types of ownership or lifetime errors are never observed because we do not mutate values marked immutable, and we do not process pointers interacting with global variables.

We resolve case 1 above by extending our configuration with the lifetime constraint the type checker could not derive from the function signature. For cases 2a and 2b, we cannot represent the reference as a safe reference because it would need multiple owners. As such, we map the relevant program location to a raw pointer in our configuration. We resolve case 2c by mapping the relevant program locations to raw pointers because of Rust's restrictions on mutable borrows. We resolve cases 2d and 2e by changing the borrowing reference to be an owned reference by finding the place the borrowed value is assigned to.

4.4 Implementation

We implemented our method by hooking into the Rust compiler to get type information and lifetime errors. We used the linter API the Rust compiler provides to access the high-level IR and type checking results, and to implement our analyses. We used the compiler driver API to run the compiler, and to collect the lifetime errors. We used `rustfix` to generate our changes to the source code of the program for each iteration.

4.5 Limitations

We limited the scope of our method to pointers not involved in external calls, global variables, pointer arithmetic, or void pointers. Besides these, our method and implementation have some other limitations.

4.5.1 Readability. The result of our conversion is not as readable as the original C program. The pervasive use of `Option` in the safe Rust program also hinders readability, as is not unifying lifetime variables that are deemed equivalent. We consider these issues to be out of scope for our prototype, as we can add later passes to reduce unnecessary lifetimes, and potentially use a non-`Option`-based reference representation if we can statically prove a pointer to be non-null.

4.5.2 Allocating objects containing unions. We do not allocate objects containing C-style unions because they lack default values. We do not arbitrarily pick one value, as this depends on the underlying representation of the union and the target architechture. Orthogonal work can convert

unions into Rust enumerations with appropriate default values, which we could then utilize in our implementation.

5 EVALUATION

We implement our tool on top of the `nightly-2020-10-15` nightly Rust compiler build version because the compiler API for Rust is not stable. We ran `c2rust` using an even older version of the compiler (the newest version that the `c2rust` developers recommend using, again due to the compiler API instability) `nightly-2019-12-05`. We run our experiments on a computer with an Intel Core i7-4790 CPU and 32 GiB RAM running Ubuntu 18.04.

5.1 Evaluation Setup

We evaluate our technique in two parts, which we label in our tables as described below:

- **ResolveImports:** This is the first step of our technique, described in Section 3.1, which resolves externally declared types and functions and removes unnecessary `unsafe` and mutability markers. Note that this step can make functions marked unsafe into safe functions even though it does not convert any raw pointers into safe references; this effect comes from removing unsafe annotations that `c2rust` adds naively when it did not need to.
- **ResolveLifetimes:** This is the remainder of our technique, described in Sections 3.2 and 3.3, which converts `Lifetime` raw pointers (as described in Section 2.3.1) into safe references. As we did in Section 3 we will use the term "raw pointers" throughout to mean specifically `Lifetime` raw pointers.

We collect the following statistics, similar to Section 2.3.1, to measure the impact of our method: the number of unsafe functions that use raw pointers; the number of raw pointer declarations; and the number of raw pointer dereferences.

5.2 Results

Table 8 shows the change in the number of unsafe functions in the scope of our method, i.e., those that are unsafe due solely to the use of `Lifetime` raw pointers as described in Section 2.3.1. Our method makes 76% of these functions safe over all of the benchmarks.

We see that ResolveImports reduces the number of unsafe functions using raw pointers by 54% even though it does not remove any raw pointers. Some of these functions did not have any underlying cause of unsafety because they use raw pointers as values (e.g., assigning them to certain fields of a struct in an initializer), which is not unsafe behavior. These cases were categorized as false positives by our definition, but making them safe requires resolving imports. ResolveLifetimes makes 46% of the remaining functions safe. The functions that are not made safe by either method were involved in the following behavior (directly or indirectly):

- Calling `free` on raw pointers that our method could not rewrite.
- Dereferencing raw pointers that our method could not rewrite.

The impact of ResolveLifetimes on making functions *completely safe* is limited because to mark a function as safe we must convert *all* dereferences of raw pointers contained in the function into dereferences of safe references. However, making half of the relevant functions safe is a good step in the right direction.

Table 9 shows the change in the declarations and dereferences of raw pointers. Overall, our method removes 87% and 89% of `Lifetime` raw pointer declarations and dereferences, respectively, over all the benchmarks. These declarations and dereferences make up 8.3% and 9.0% of the *total* number of raw pointer declarations and dereferences including all categories of unsafety, because

Table 8. Number of unsafe functions due uniquely to using raw pointers. ResolveImports and ResolveLifetimes are the two phases of our method explained in Section 5.1; the corresponding columns show how many formerly unsafe functions were made safe by each phase (remembering that ResolveLifetimes is executed on the result of ResolveImports).

Benchmark	Original	ResolveImports	ResolveLifetimes	Remaining	Total made safe (%)
libcsv	12	0	11	1	92%
urlparser	2	0	0	2	0%
RFK	2	1	0	1	50%
genann	3	2	0	1	67%
lil	6	2	1	3	50%
json-c	20	9	8	3	85%
bzip2	4	0	4	0	100%
libzahl	2	0	2	0	100%
TI	45	44	0	1	98%
tinycc	35	6	2	4	89%
optipng	62	32	9	21	66%
libxml2	36	27	9	18	50%
Total	229	123	46	55	76%

Table 9. Number of raw pointer declarations and dereferences. Orig. = The number from the original program. Fixed = The number of raw pointer declarations or dereferences removed by our method.

Benchmark	Raw Ptr. Declarations				Raw Ptr. Dereferences			
	Orig.	Remaining	Fixed	Fixed (%)	Orig.	Remaining	Fixed	Fixed (%)
libcsv	18	0	18	100%	148	0	148	100%
urlparser	5	0	5	100%	58	0	58	100%
RFK	1	0	1	100%	0	0	0	–
genann	5	0	5	100%	0	0	0	–
lil	50	27	23	46%	636	22	614	97%
json-c	41	10	31	76%	93	16	77	83%
bzip2	37	8	29	78%	679	599	80	12%
libzahl	7	0	7	100%	31	0	31	100%
TI	82	82	0	0%	0	0	0	–
tinycc	192	1	191	99%	946	42	904	96%
optipng	210	10	200	95%	606	16	590	97%
libxml2	800	47	753	94%	4472	120	4352	97%
Total	1448	185	1263	87%	7669	815	6854	89%

of the multi-faceted nature of how raw pointers are used. Three of our benchmarks (RFK, genann, and TI) do not dereference any `Lifetime` raw pointers, so they do not get much improvement from our method. We investigated the declarations and dereferences that our method fails to remove. They fall under the following categories:

- The pointer is not used safely according to the borrow checker rules. This is the case for the pointers in `libxml2`, `optipng`, and `bzip2` that we fail to remove, and one declaration in `json-c`.
- The pointer is used in the signature of a function that is used as a function pointer. This is the case for the pointers in `json-c` (on all but one declaration we failed to remove), `lil`, and `TI` that we fail to remove.

Table 10. Run time performance results of our method. ResolveImports and ResolveLifetimes are explained in Section 5.1. "iter." stands for iteration. # iter. is the number of iterations of ResolveLifetimes before we reach a program with no lifetime errors. t_{iter} is the total time ResolveLifetimes takes per iteration. $t_{analysis/iter}$ is the time ResolveLifetimes spends in taint analyses described in Section 3.3.

Benchmark	ResolveImports	ResolveLifetimes				Total time (s)
	Time (s)	# iter.	t_{iter} (s)	$t_{analysis/iter}$ (s)	Total time (s)	
libcsv	0.3	2	0.5	0.3	1.0	1.3
urlparser	0.4	2	0.6	0.4	1.2	1.6
RFK	0.4	1	0.6	0.4	0.6	1.0
genann	0.6	1	1.0	0.6	1.0	1.6
lil	0.9	1	2.2	1.2	2.2	3.1
json-c	1.2	2	2.4	1.4	4.8	6.0
bzip2	2.4	2	4.2	2.7	8.4	10.8
libzahl	1.5	1	2.3	1.7	2.3	3.8
TI	2.2	1	4.8	2.1	4.8	7.0
tinycc	9.7	2	13.3	14.8	26.6	36.3
optipng	7.4	3	19.0	12.2	57.0	64.4
libxml2	40.5	5	334.3	321.0	1671.5	1712.0

The other reason for failing to convert some raw pointers is a limitation of our method in that we do not rewrite function pointer types, so we cannot change the signature of the functions passed to function pointers. We also inspected the intermediate steps of our tool to look into the root causes related to the pointers that remain raw due to borrow checker violations. In the `bzip2` and `optipng` benchmarks, violating the borrow checker for one pointer (in `bzip2`) and two pointers (in `optipng`) are the reason for all of the raw pointers that remain after our technique; in both benchmarks, the pointer value with illegal borrowing flows into a struct field, so any use of that struct field also becomes a raw pointer.

5.2.1 ***Limitations of ResolveImports.*** The core assumption of our heuristics for ResolveImports is that the structs with the same name and the same fields represent the same data type, so their definitions can be merged to allow importing functions from other modules in the same program. This assumption is violated in the `tinycc` benchmark for four anonymous structs, because the `c2rust`-generated names of those structs did not match across modules because of how `c2rust` generates names for anonymous structs. Because of this problem we get an error from the Rust compiler after the ResolveImports phase, and fixing the issue involved importing the four structs from where they are defined, removing the duplicate definition, and changing the four lines of code that use them. The fix was a 38-line patch, and it took one of the authors 10 minutes to investigate and fix the issue.

5.3 Performance

Running our method is a one-time effort when translating the C program to a Rust program. Table 10 lists the run time performance. In all of our benchmarks except `libxml2` and `optipng` our method finishes under a minute. In all benchmarks, ResolveLifetimes takes the majority of the time (harmonic mean: 72.8%). In all benchmarks except `libxml2` ResolveLifetimes takes at most 3 iterations to resolve all borrow checker conflicts. On `libxml2` our method takes 29 minutes to finish. 94% of this time is due to the taint analyses we perform to propagate the information on which locations need to be owned references or raw pointers as described in Section 3.3; the taint analysis implementation is not very optimized and can be improved.

6 RELATED WORK

This section covers relevant background regarding C to Rust translation and other related work. Rust's ownership system and memory management method are reviewed in Appendix A for unfamiliar readers.

6.1 Translating C to Rust

There have been several early tools for translating C code to unsafe Rust code, such as Citrus [Citrus Developers [n.d.]] and Corrode [Sharp 2020]. Both of these tools are now superseded by `c2rust` [Immunant inc. 2020b], an industry-backed C99 to Rust translator. It is made of three parts: (1) the *translator* translates the C program to an unsafe Rust program that mirrors the C code; (2) the *refactoring tool* helps the programmer refactor and rewrite the unsafe Rust program into a mostly-safe Rust program by providing program-wide refactoring operations and scripting support; and (3) the *cross-check tool* allows for comparing the execution traces of two programs on a test input. Since `c2rust` does not have any formal guarantees, it relies on the cross-check tool to validate that the initial Rust program behaves the same as the original C program and that the incrementally refactored Rust programs preserve that behavior. Our technique's implementation leverages the translator tool to provide an initial unsafe Rust program and the cross-check tool to validate that the output of our technique behaves the same as the original C program.

6.2 Characterizing Unsafe Code in Rust

Astrauskas et al. [2020] investigate usage of `unsafe` in practice across the open-source Rust ecosystem. They use manual inspection and automated queries to analyze program structure, types, and other information produced by the compiler. They find that most unsafe code is simple and well-encapsulated, however interoperability with other languages causes unsafe features to be used extensively. They report that 44.6% of the unsafe function definitions they found in the Rust ecosystem are bindings for foreign functions used for linking against C libraries. Their results support that porting these C libraries to safer Rust versions would significantly reduce the overall amount of unsafe dependencies in the Rust ecosystem.

Qin et al. [2020] empirically investigate the usage of Rust's safety mechanisms and `unsafe` in open source Rust projects. They also build two static bug detectors based on their study results, and revealed previously unknown bugs. Their results show that 66% of unsafe operations are due to unsafe memory operations such as type casting and raw pointer manipulation. They also report that the most common (42%) purpose of `unsafe` usage is to reuse existing code, including C code that performs pointer manipulation and calling into external libraries like `glibc`. These results indicate that converting C code to safe Rust is an important problem to increase trust in Rust code, and that converting raw pointer operations to safe Rust references accounts for a significant portion of this conversion.

6.3 Formalizing Rust Ownership and Type Systems

There are several Rust formalizations in the literature [Benitez 2016; Jung et al. 2017; Reed 2015; Weiss et al. [n.d.]]. Here, we cover the formalizations that involve the Rust ownership system or borrow checker, as our technique interacts with both components.

Patina [Reed 2015] is a formal semantics for a pre-1.0 version of Rust. It focuses on using a syntactic version of the borrow checking algorithm based on lexically-scoped lifetimes. Since then, Rust has added support for non-lexical lifetimes and other new features, making safety now less restrictive than in pre-1.0 Rust.

The RustBelt project [Jung et al. 2017] describes a mechanised formal semantics for a Rust mid-level intermediate representation (MIR) called λ_{Rust}. λ_{Rust} has been used to derive the verification conditions for safety of widely-used standard library abstractions using `unsafe`, and to formally prove that the API they expose is a safe extension of the language. λ_{Rust} includes a complete Rust specification. However, since our technique requires reasoning only about Rust programs translated from C which do not use all Rust features (e.g., traits), we do not need a complete Rust specification. Therefore, we decided to base our work on Oxide [Weiss et al. [n.d.]], a simpler formalization that operates closer to Rust's source level and only involves Rust features observed in our input programs. Oxide handles explicit mutability and lifetime annotations with the aim of capturing *the essence of Rust*. Oxide is close to Rust's high-level IR (HIR), and does not model Rust's module or trait systems.

There have also been extensions of existing semantic modeling and verification tools to support Rust. Baranowski et al. [2018] extend the SMACK verifier to work on Rust programs, and KRust [Wang et al. 2018] is an implementation of Rust's semantics on the K-framework.

7 CONCLUSION

In this paper we have investigated the problem of automatically translating C programs into *safer* Rust programs—that is, Rust programs that improve on the safety guarantees of the original C programs. First, we conducted an in-depth study into the underlying causes of unsafety in translated programs and the relative impact of fixing each cause. We find that there is a relatively small set of well-defined categories for these causes; however, the majority of unsafety in a translated program is caused by more than one category. This means that fixing any one category will have only a small impact, and that fixing a majority of unsafety will require addressing multiple categories. We have ordered the categories by their impact to help determine their relative priorities.

Second, we have described and evaluated a novel technique for automatically removing a particular category of unsafety: the `Lifetime` raw pointers. Our technique piggy-backs on the Rust compiler, and our evaluation shows that it removes 87% of `Lifetime` raw pointer declarations and 89% of raw pointer dereferences of this category.

This paper presents the first empirical study of unsafety in *translated* Rust programs (as opposed to programs originally written in Rust) and also the first technique for automatically removing causes of unsafety in translated Rust programs. It lays the groundwork for future research into removing even more unsafety from these programs. That future research will address the other categories of unsafety outlined in this paper and ultimately extend the project to handle multi-threaded programs and C++.

REFERENCES

[n.d.]a. Rust support hits linux-next. https://lwn.net/Articles/849849/

[n.d.]b. Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/

2021a. NVD - CVE-2021-21148. https://nvd.nist.gov/vuln/detail/CVE-2021-21148

2021b. NVD - CVE-2021-3156. https://nvd.nist.gov/vuln/detail/CVE-2021-3156

Brian Anderson, Lars Bergstrom, David Herman, Josh Matthews, Keegan McAllister, Manish Goregaokar, Jack Moffitt, and Simon Sapin. 2015. Experience Report: Developing the Servo Web Browser Engine using Rust. *arXiv:1505.07383 [cs]* (May 2015). http://arxiv.org/abs/1505.07383 arXiv: 1505.07383.

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428204

Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust programs with SMACK. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 528–535.

Sergio Benitez. 2016. Short Paper: Rusty Types for Solid Safety. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/2993600.2993604

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 211–230. https://doi.org/10.1145/582419.582440

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. 2003. Ownership Types for Safe Region-based Memory Management in Real-time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 324–337. https://doi.org/10.1145/781131.781168

David Bryant. 2016. A Quantum Leap for the Web. https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12

Citrus Developers. [n.d.]. Citrus / Citrus. https://gitlab.com/citrus-rs/citrus

Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

Tim Hutt. 2021. Would Rust secure cURL? https://timmmm.github.io/curl-vulnerabilities-rust/

Immunant inc. 2020a. c2rust Manual Examples. https://c2rust.com/manual/examples/index.html

Immunant inc. 2020b. immunant/c2rust. https://github.com/immunant/c2rust original-date: 2018-04-20T00:05:50Z.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

S. Klabnik and C. Nichols. 2018. *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/

Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/2818302.2818306

Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2016. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management - ISMM 2016*. ACM Press, Santa Barbara, CA, USA, 89–98. https://doi.org/10.1145/2926697.2926707

Nicholas D Matsakis. 2018. An alias-based formulation of the borrow checker. https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. (2020), 763–779. https://doi.org/10.1145/3385412.3386036

Eric Reed. 2015. *Patina: A formalization of the Rust programming language*. Master's thesis. University of Washington Department of Computer Science and Engineering.

Jamey Sharp. 2020. jameysharp/corrode. https://github.com/jameysharp/corrode original-date: 2016-05-05T21:12:52Z.

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Jeff Vander Stoep and Stephen Hines. 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html

The Rust developers. [n.d.]a. std::option - Rust standard library documentation. https://doc.rust-lang.org/std/option/index.html#representation

The Rust developers. [n.d.]b. The Rust Reference. https://doc.rust-lang.org/stable/reference/

F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 44–51. https://doi.org/10.1109/TASE.2018.00014

Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. [n.d.]. Oxide: The Essence of Rust. ([n. d.]), 27.

A APPENDIX: RUST'S OWNERSHIP SYSTEM

This appendix serves a short primer to how Rust handles *ownership* and *borrowing*. Both of these features are central to Rust's memory model, and enable it to statically ensure memory safety in safe code without resorting to garbage collection at runtime. Given that our work must work with Rust's memory model closely, it is necessary to have some understanding of Rust's memory model in order to understand the significance of our own work. That said, this appendix is intended only as a quick introduction; readers curious for more details are directed to the online Rust book for

basics [Klabnik and Nichols 2018], as well as as a more formal alias-based formulation at [Matsakis 2018].

A.1 Motivation

Rust's memory model ensures memory safety statically, without resorting to potentially expensive runtime memory management techniques like garbage collection. In Rust, well-typed programs are memory-safe by construction. As with a garbage collected language, users explicitly perform memory allocation, but do not explicitly perform deallocation. Unlike with garbage collection, the Rust compiler statically inserts routines to deallocate heap-allocated memory when it is no longer needed. The type system of Rust is designed in such a manner that the compiler statically knows exactly where these memory deallocations need to be performed. This knowledge of when to perform deallocation is based around *ownership*.

A.2 Ownership

By default, data is said to be *owned* in Rust. For example, consider the following function definition `f`, which uses type `Vec` from the Rust standard library (representing a vector):

```
1 fn f(v: Vec<i32>) {}
```

`f` is said to take ownership of `v`. This is indicated by the fact that `v` is directly of type `Vec<i32>`. Whoever owns the data is ultimately responsible for deallocating any heap-allocated data held. Deallocation implicitly occurs whenever the variable bound to the data falls out of scope. With this in mind, any heap-allocated data held in `v` is deallocated immediately after the call to `f`, as `v` will no longer be accessible.

Within a scope, ownership can be transferred from one variable to another. For example, consider the following code snippet:

```
1 fn example() {
2   let v1 = vec![1, 2, 3]; // creates a vector holding 1, 2, 3
3   let v2 = v1;
4 }
```

In this case, `v1` initially holds the underlying vector. Ownership is then transferred to variable `v2`. Because ownership is never transferred away from `v2`, `v2` will have all heap-allocated memory deallocated at `example`'s termination. Because ownership was transferred away from `v1`, there is no similar deallocation performed for `v1`, beyond typical stack deallocation of `v1`.

Ownership can also be transferred between scopes. For example, consider the following:

```
1 fn identity(v: Vec<i32>) -> Vec<i32> { return v; }
```

In this case, like the prior `f` example, `identity` takes ownership over `v`. However, because `identity` later returns `v`, it transfers ownership to `identity`'s caller. Any heap-allocated memory bound to `v` then becomes the concern of `identity`'s caller.

A.3 Borrowing and Lifetimes

While the ownership model unambiguously allows the compiler to safely statically deallocate all heap-allocated memory, it is nonetheless very restrictive. For example, if you wanted to define a function that merely printed the contents of a vector, it would need to transfer ownership back to the caller. This would mean having an unintuitive type signature like:

```
1 fn print_all(v: Vec<i32>) -> Vec<i32> { ... }
```

With this in mind, the more data a function needs to do its job, the more data the very same function needs to return. There are also negative performance implications of ownership transfer, since barring compiler optimizations, it entails copying any stack-allocated memory behind a variable.

To address these issues around ownership transfer, Rust also has a concept known as *borrowing*. As the name suggests, data can be temporarily borrowed without changing ownership. Data is borrowed through a reference, which bear similarity to references in other languages. Borrowed data can be used like owned data, with some restrictions. One important restriction is that borrowed data cannot outlive the actual data being borrowed. Using C/C++ terminology, Rust must ensure that there are no dangling pointers to any allocated data.

To ensure that the underlying data being borrowed is always valid, Rust introduces the concept of a *lifetime*. Lifetimes are type-level variables which abstractly define how long the underlying data being borrowed will be in memory. For example, consider the following code:

```
fn has_lifetime<'a>(v: &'a Vec<i32>) { ... }
```

Instead of having ownership of `v` transferred to `has_lifetime`, this instead borrows the underlying `Vec<i32>` for lifetime `'a`. Rust will ensure that the underlying `Vec<i32>` is in memory for the duration of the call to `has_lifetime`. Because `has_lifetime` merely borrows the `Vec<i32>`, there is no memory deallocation of `v` performed; `has_lifetime` does not own the vector, and so it is not `has_lifetime`'s responsibility to deallocate the vector.

Like regular type variables, data structure definitions themselves can take lifetimes, as with:

```
struct SomeData<'a, 'b> {
  first: &'a i32,
  second: &'b i32
}
```

With the above code in mind, Rust will make sure that no allocated instance of `SomeData` will outlive anything it borrows. That is, the data referred to by `first` and `second` will always be in memory at least as long as the `SomeData` data structure itself.

To show this in practice, consider the following example, which is rejected by the Rust compiler:

```
fn rejected() {
  let the_data;
  let first_int = 1;
  {
    let second_int = 2;
    the_data = SomeData { first: &first_int, second: &second_int };
  }
  print!("{}'', *the_data.second);
}
```

The above code is rejected by the Rust compiler, with an error message stating that `second_int` does not live long enough. To understand why, first understand that each block in Rust corresponds to a separate lifetime variable. That is, an enclosing scope maps directly to object lifetimes. For speaking purposes, the outer scope of rejected will be called `'a`, and the inner scope (where `second_int` is declared) will be called `'b`. With this in mind, `the_data` has type `SomeData<'a, 'b>`, and it itself has lifetime `'a`. However, `'b` does not live as long as `'a`. As such, we have attempted to create a data structure with a lifetime longer than its constituents, which is not permitted. As such, Rust rejects the program. Thinking in terms of C/C++, this rejection makes sense - `second_int` is allocated on the stack and subsequently deallocated after `the_data` is initialized, so `the_data.second` would be a dangling pointer.

A.3.1 Restrictions. All borrows seen so far are immutable borrows, meaning that the underlying object cannot be changed through these borrows. Furthermore, the underlying object may not be changed at all while any immutable borrows are active. Similarly, Rust disallows ownership transfers while any borrows are active. This can be statically checked at compile time, as shown in the code below:

```
1 struct MyStruct {
2     first: i32
3 }
4
5 fn involves_borrows<'a>(datum: &'a MyStruct) -> &'a MyStruct {
6   return datum;
7 }
8 fn performs_transfer(x: MyStruct) {}
9
10 fn main() {
11     let x = MyStruct { first: 42 };
12     let r = involves_borrows(&x);
13     performs_transfer(x);
14     print!("{}", r.first)
15 }
```

The above code fails to compile, as the the transfer performed by `performs_transfer` is disallowed because reference `r` still refers to the same data structure. Specifically, Rust tracks that `x` has an active borrow at the call to `performs_transfer`, disallowing the call. As an aside, the subsequent use of `r.first` is required to get this code to compile, as this forces the compiler to internally keep the borrow of `x` around after the call to `performs_transfer`; effectively, Rust will permit the existence of a dangling pointer, but not the access of a dangling pointer.

A.3.2 Immutable and Mutable Borrows. All prior borrow examples are based on immutable borrows, meaning the underlying object cannot be changed through the borrow. Rust also supports mutable borrows, which use the `mut` reserved word, like so:

```
&'a mut Vec<i32>
```

The above snippet refers to a mutable borrow of a `Vec<i32>`, where the underlying vector is in memory for at least `'a` lifetime.

Mutable borrows work similarly to mutable borrows, with the following twists. With immutable borrows, the same data may be borrowed multiple times in the same context, as none of the borrows can change the underlying object. However, with mutable borrows, only one such mutable borrow may be active at any time. Furthermore, if a mutable borrow is active, all mutation must be done through the mutable borrow, and no immutable borrows or ownership transfers are permitted. While restrictive, these requirements prevent data races from occurring - all mutation is very carefully tracked and made explicit in the types; it is not possible for data to be modified "out from under you", as it is in most languages.